Evaluating the performance of model transformation styles in Maude*

Roberto Bruni[1] and Alberto Lluch Lafuente[2]

[1] Department of Computer Science, University of Pisa, Italy
[2] IMT Institute for Advanced Studies Lucca, Italy

Abstract. Rule-based programming has been shown to be very successful in many application areas. Two prominent examples are the specification of model transformations in model driven development approaches and the definition of structured operational semantics of formal languages. General rewriting frameworks such as Maude are flexible enough to allow the programmer to adopt and mix various rule styles. The choice between styles can be biased by the programmer's background. For instance, experts in visual formalisms might prefer graph-rewriting styles, while experts in semantics might prefer structurally inductive rules. This paper evaluates the performance of different rule styles on a significant benchmark taken from the literature on model transformation. Depending on the actual transformation being carried out, our results show that different rule styles can offer drastically different performances. We point out the situations from which each rule style benefits to offer a valuable set of hints for choosing one style over the other.

1 Introduction

Many engineering activities are devoted to manipulate software artifacts to enhance or customize them, or to define their possible ordinary evolutions and exceptional reconfigurations. The concept of *model* as unifying software artifact representation has been promoted as a means to facilitate the specification of such activities in a generic way. Many dynamic aspects can be conceived as *model transformations*: e.g. architectural reconfigurations, component adaptations, software refactorings, and language translations. Rule-based specifications have been widely adopted as a declarative approach to enact model-driven transformations, thanks to the solid foundations offered by rule-based machineries like term [1] and graph rewriting [2].

Recently we have investigated the possibility to exploit the structure of models to enhance software description and to facilitate model transformations [3, 4]. Indeed, many domains exhibit an inherently hierarchical structure that can be exploited conveniently to guarantee scalability. We mention, among others, nested components in software architectures and reflective object-oriented systems, nested sessions and transactions in business processes, nested membranes

* Work supported by the EU Project ASCENS and the Italian MIUR Project IPODS.

in computational biology, composition associations in UML-like modeling frameworks, semi-structured data in XML-like formats, and so on. Very often such layering is represented in a plain manner by overlapping the intra- and the inter-layered structure. For instance, models are usually formalised as flat object configurations (e.g. graphs) and their manipulation is studied with tools and techniques based on rewriting theories that do not fully exploit the hierarchical structure. On the other hand, an *explicit* treatment of the hierarchical structure for specifying and transforming model-based software artifacts is possible. As a matter of fact, some layering structures (like composition relations in UML-like languages) can be conveniently represented by an explicit hierarchical structure enabling then hierarchical manipulations of the resulting models (see e.g. [3, 4]).

We have investigated such issues in previous work [3] proposing an approach analogous to the *russian dolls* of [5, 6], where objects can be nested within other objects. In this view, *structured* models are represented by terms that can be manipulated by means of term-rewrite techniques like conditional term rewriting [1]. In [3] we compared the flat representation against the nested one, showing that they are essentially equivalent in the sense that one can bijectively pass from one to the other. Each representation naturally calls for different rule styles and the comparison in [3] mainly addressed methodological aspects, leaving one pragmatical issue open: how to decide in advance which approach is more efficient for actually executing a model transformation?

We offer an answer to this question in this paper. We have selected two prominent approaches to model transformation. The first one is archetypal of the graph-transformation based model-driven community and follows the style of [7]. The second one is quite common in process calculi and goes along the tradition of Plotkin's structural operational semantics, as outlined in [3]. Both approaches can be adopted in flexible rule-based languages like Maude [8] (the rewriting logic based language and framework we have chosen). In order to obtain significant results we have implemented three test cases widely used in the literature: the reconfiguration of components that migrate from one location to another one, the transformation of class diagrams into relational schemas, and the refactoring of class diagrams by pulling up attributes. As a byproduct we offer a novel implementation of these three classical transformations based on conditional rules. Indeed, such style of programming model transformations has not been proposed by other authors, as far as we know.

Our experimental results stress the importance of choosing the right transformation style carefully to obtain the best possible performance. We point out some features of the examples that impact the performance of each rule format, thus providing the programmer with a set of valuable guidelines for programming model transformations in expressive rule-based frameworks like Maude.

Synopsis. § 2 presents a graph-based algebraic representation of models as nested object collections and describes rewrite rule styles for implementing model transformations in Maude. § 3 presents some enhancements that can be applied to the transformation styles. § 4 describes our benchmark. § 5 presents the experimental results. § 6 concludes the paper.

2 Preliminaries

In this section we illustrate the two key model transformation paradigms and the Maude notation we shall exploit in the rest of the paper over a basic example of transformation, namely from trees to list. A classical approach would provide ad-hoc data structures for trees and lists and an ad-hoc algorithm for implementing the transformation. Model driven approaches, instead, consider a common representation formalism for both data structures and a generic transformation procedure that acts on such formalism. In our setting, the representation formalism for models are collections of attributed objects and the transformation procedure is based on rewrite rules.

The Maude language already provides some machinery for this purpose, called object-based configurations [8], which we tend to follow with slight modifications aimed to ease the presentation. More precisely we represent models as *nested object collections* [3] (following an idea originally proposed in [5] and initially sketched in [6]), which can be understood as a particular class of attributed, hierarchical graphs. We then implement transformations as sets of rewrite rules.

Rewriting Logic and Maude. Maude modules describe theories of rewriting logic [1], which are tuples $\langle \Sigma, E, R \rangle$ where Σ is a signature, specifying the basic syntax (function symbols) and type machinery (sorts, kinds and subsorting) for terms, e.g. model descriptions; E is a set of (possibly conditional) equations, which induce equivalence classes of terms, and (possibly conditional) membership predicates, which refine the typing information; R is a set of (possibly conditional) rules, e.g. model transformations.

The signature Σ and the equations E of a rewrite theory form a *membership equational theory* $\langle \Sigma, E \rangle$, whose initial algebra is denoted by $T_{\Sigma/E}$. Indeed, $T_{\Sigma/E}$ is the state space of a rewrite theory, i.e. states (e.g. models) are equivalence classes of Σ-terms modulo the least congruence induced by the axioms in E (denoted by $[t]_E$ or t for short). Sort declarations takes the form `sort S` and subsorting (i.e. subtyping) is written `subsort S < T`. For instance, the sort of objects (`Obj`) is a subsort of configurations `sort Configuration` as declared by `subsort Obj < Configuration`.

Operators are declared in Maude notation as `op f : TL -> T [As]` where `f` is the operator symbol (possibly with mixfix notation where underscores `_` stand for argument placeholders), `TL` is a (possibly empty, blank separated) list of domain sorts, `T` is the sort of the co-domain, and `As` is a set of equational attributes (e.g. associativity, commutativity). For example, object configurations (sort `Configuration`) are constructed with operators for the empty configuration (`none: -> Configuration`), single objects (via subsorting) or the union of configurations, denoted with juxtaposition (`_ _ : Configuration Configuration -> Configuration [assoc comm id:none]`), declared to be associative, commutative and to have `none` as its identity operator (i.e. they are multisets).

Each object represents an entity and its properties. Technically, an object is defined by its identifier (of sort `Oid`), its class (of sort `Cid`) and its attributes (of

sort `AttSet`). Objects are built with an operation `< _ : _ | _ >` with functional type `Oid Cid AttSet -> Obj`. Following Maude conventions, we shall use quoted identifiers like `'a` as object identifiers, while class identifiers will be defined by ad-hoc constructors. In our running example we use the constants `Node` and `Item` of sort `Cid` to denote the classes of tree nodes and list items, respectively.

The attributes of an object define its properties and relations to other objects. They are basically of two kinds: datatype attributes and association ends. Datatype attributes take the form `n: v`, where `n` is the attribute name and `v` is the attribute value. For instance, in our running example we shall consider a natural attribute `value` (sort `Nat`), representing the value of a node or item. A node with identifier `'a` and value `5` is denoted by `< 'a : Node | value: 5 >`.

Relations between objects can be represented in different ways. One typical approach is to use a pair of references (called *association ends* in UML terminology) for each relation. So if an object `o1` is in relation `R` with object `o2` then `o1` is equipped with a reference to `o2` and vice versa. In our case this is achieved with attributes of the form `R: O2` and `opp(R): O1` where `R` indicates the relation name and `O1`, `O2` are sets of object identifiers (sort `OidSet`). Association ends of the same relation within one object are grouped together (hence the use of identifier sets as domain of association attributes). In our example we have two relations `left` and `right` between a node and its left and right children, and one relation `next` between an item of the list and the next one. Clearly, the opposite relations of `left`, `right` and `next` are the parent and previous relations. As an example of a pair of references consider a node `< 'a : Node | value: 5 , left: 'b >` and its son `< 'b : Node | value: 3 , opp(left): 'a >`. Of course an object can be equipped with any number of attributes. Actually, the attributes of an object form a set built out of singleton attributes, the empty set (`none`) and union set (denoted with `_ , _`).

The following simple configuration represents a tree with three nodes.

```
< 'a : Node | value: 5 , left: 'b ,  right: 'c >
< 'b : Node | value: 3 , opp(left): 'a >
< 'c : Node | value: 7 , opp(right): 'a >
```

Operation `<< _ >> : Configuration -> Model` wraps a configuration into a model.

Functions (and equations that cannot be declared as equational attributes) are defined by a set of confluent and terminating conditional equations of the form `ceq t = t' if c`, where where `t`, `t'` are Σ-terms, and `c` is an application condition. When the application condition is vacuous, the simpler syntax `eq t = t'` can be used. For example, an operator `op size : Configuration -> Nat` for measuring the number of objects in a configuration is inductively defined by equations `eq size(none) = 0` and `eq size(O C) = 1 + size(C)` (with `O`, `C` being variables of sort `Obj`, `Configuration`, respectively). Roughly, an equational rule can be applied to a term `t''` if we find a match `m` (i.e. a variable substitution) for `t` at some place in `t''` such that `m(c)` holds (i.e. `c` after the application of the substitution `m` evaluates to true). The effect is that of substituting the matched part with `m(t')`. For example, calculating the size of the above tree is done by

reducing `size(< 'a : Node | value: 5 , left: 'b , right: 'c > < 'b : Node | value: 3 , opp(left): 'a > < 'c : Node | value: 7 , opp(right): 'a >)` to `1 + size(< 'b : Node | value: 3 , opp(left): 'a > < 'c : Node | value: 7 , opp(right): 'a >)`, then to `2 + size(< 'c : Node | value: 7 , opp(right): 'a >)` and finally to 3.

Structured models. A *nested object collection* allows objects to have *container attributes*, i.e. configuration domain attributes. While in a plain object collection a containment relation `r` between two objects `o1` and `o2` is represented by exploiting a pair of association end attributes `r` and `opp(r)`, now `o2` is embedded into `o1` by means of the container attribute `r`. For instance, the above tree becomes

```
< 'a : Node | value: 5 ,
              left: < 'b : Node | value: 3 > ,
              right: < 'c : Node | value: 7 > >
```

The hierarchical structure of models forms a tree. The two approaches that we have described differ essentially in the way we represent such a tree. Indeed, flat and nested representations are in bijective correspondence, i.e. for each flat object collection we can obtain a unique nested collection and vice versa as shown in [3], so that we can pass from one to the other as we find more convenient for specific applications or analyses.

Transformations as sets of rewrite rules. Transformations can be defined by means of rewrite rules, which take the form `crl t => t' if c`, where `t`, `t'` are Σ-terms, and `c` is an application condition (a predicate on the terms involved in the rewrite, further rewrites whose result can be reused, membership predicates, etc.). When the application condition is vacuous, the simpler syntax `rl t => t'` can be used. Matching and rule application are similar to the case of equations with the main difference being that rules are not required to be confluent and terminating (as they represent possibly non-deterministic concurrent actions rather than functions). Equational simplification has precedence over rule application in order to simulate rule application modulo equational equivalence.

SPO transformations. The need for visual modelling languages and the graph-based nature of models have contributed to the success of graph transformation approaches to model transformations. In such approaches, transformations are programmed in a declarative way by means of a set of graph rewrite rules. The transformation style that we consider here is based on the algebraic graph transformation approach [2]. The main idea is that each rule has a left-hand side and a right-hand side pattern. Each pattern is composed by a set of objects (nodes) possibly interrelated by means of association ends (edges). A rule can be applied to a model whenever the left-hand side can be matched with part of the model, i.e. each object in the left-hand side is (injectively) identified with an object and idem for the association ends. The application of a rule removes the matched part of the model that does not have a counterpart in the right-hand side and, vice versa, adds to the model a fresh copy of the right-hand side part

that is not present in the left-hand side. Items in common between the left-hand side and the right-hand side are preserved during the application of the rule. Very often, rules are equipped with additional application conditions, including those typical of graph transformation systems (e.g. to avoid dangling edges) and its extensions like *Negative Application Conditions* (NACs).

In our setting, this means that rules have in general the following format:

```
crl << lhs conf1 >> => << rhs conf1 >> if applicable(lhs conf1) .
```

where `lhs` and `rhs` stand for the rule's left- and right-hand side configurations, `conf1` as the context in which the rule will be applied, and `applicable` is the boolean function implementing the application condition. Simpler forms are possible, e.g. in absence of application conditions the context is not necessary and rules take the form: `rl lhs => rhs .`

In our running example the transformation rules basically take a node `x` and its children `y` and `z` and puts them in some sequence, with `x` before `y` and `z`. This rule might introduce branches in the sequence that are solved by appropriate rules. A couple of rules are needed to handle some special cases, like `x` being the root or a node that has already been put in the list (in the middle, tail or head). Let us show one of the basic rules (the rest of the rules are very similar):

```
rl [nodeToItem]
   << < x : Node | value: vx , left: y , right: z , next: u , Ax >
      < y : Node | value: vy , op(left): x, Ay >
      < z : Node | value: vz , op(right): x, Az >
      < u : Node | value: vu , op(next): x, Au >
      conf1 >> =>
   << < x : Item | value: vx , next: y , Ax>
      < y : Node | value: vy , op(next): x, next: z , Ay >
      < z : Node | value: vz , op(next): y, next: u , Ay >
      < u : Node | value: vu , op(next): z, Au >
      conf1 >> .
```

SOS transformations. We now describe transformation rules in the style of *Structural Operational Semantics* [9] (SOS). The basic idea is to define a model transformation by structural induction, which in our setting basically amounts to exploiting set union and (possibly) nesting.

We recall that SOS rules make use of labels to coordinate rule applications. We first present the implementation style of SOS semantics in rewriting logic as described in [10] and then present our own encoding of SOS which provides a more efficient implementation, though circumscribed to some special cases.

The approach of [10] requires to enrich the signatures with sorts for rule labels (`Lab`), label-prefixed configurations `LabConfiguration`, and a constructor `{_}_ : Lab Configuration => LabConfiguration` for label-prefixed configurations. In addition, rule application is allowed at the top-level of terms only (via Maude's `frozen` attribute [11]) so that sub-terms are rewritten only when required in the premise of a rule (as required by the semantics of SOS rules). With this

notation a term {lab1}conf1 represents that a configuration conf1 that has been obtained after application of a lab1-labelled rule.

One typical rule format in our case allows us to conclude a transformation lab1 for a configuration made of two parts conf1 and conf2 provided that each part can respectively perform some transformation lab2, lab3:

```
crl conf1 conf2 => {lab3} conf3 conf4
 if conf1 => {lab1} conf3
 /\ conf2 => {lab2} conf4 .
```

Typically, the combination of labels will follow some classical form. For instance, with Milner-like synchronisation, lab1, lab2 can be complementary actions, in which case lab3 would be a silent action label. Instead, Hoare-like synchronisation would require lab1, lab2 and lab3 to be equal.

Consider now a hierarchical representation of models based on nested object collections. In this situation we need rules for dealing with nesting. Typically, the needed rule format is the one that defines the transformation lab1 of an object oid1 conditional to some transformation lab2 of one of its contents c:

```
crl < oid1 : cid1 | c: conf1 , attSet1 > =>
    {lab1} < oid1 : cid1 | c: conf2 , f(attSet1) >
 if conf1 => {lab2} conf2 .
```

Such rules might affect the attributes of the container object (denoted with function f) but will typically not change the object's identifier or class. More elaborated versions of the above rule are also possible, for instance involving more than one object or not requiring any rewrite of contained objects.

In our running example we have the following rule that transforms a tree provided that its subtrees can be transformed into lists

```
crl [root] : < x : Node | value: vx , left: leftTree , right: rightTree >
          => {toList}
             list1
             < tail : Item | value: vt, next: x , opp(next): y >
             < x : Item | value: vx , opp(next): tail , next: head >
             < head : Item | value: vh, opp(next): x , next: z >
             list2
 if leftTree => {toList} list1 < tail : Item | value: vt , opp(next): y >
 /\ rightTree => {toList}  < head : Item | value: vh , next: z > list2 .
```

Note that head and tail of the transformed sublists are identified by the lack of next and opp(next) attributes. Rules are also needed to handle leafs:

```
rl [root] : < x : Node | value: vx > => {toList} < x : Node | value: vx > .
```

Finally, rules are needed to close the transformations at the level of models. Such rules have the following format:

```
crl << conf1 >> => << conf2 >> if conf1 => {lab1} conf2 .
```

In our example the rule would be

```
crl << conf1 >> => << conf2 >> if conf1 => {toList} conf2 .
```

3 Enhanced SOS implementation

While performing our preliminary experiments we discovered a more efficient way to encode SOS rules in rewriting logic that we call SOS*.

The most significant improvement applies to those cases in which the labels of the sub-configurations are known in advance. As a matter of fact this was the case of all test cases we consider in the next section. The idea is to put the labels on the left-hand side of rules as a sort of context requiring the firing of transformations with such label.

As a more general example the above rule scheme becomes now:

```
crl {lab3} conf1 conf2 => conf3 conf4
 if {lab1} conf1 => conf3
 /\ {lab2} conf2 => conf4 .
```

The main difference is that now `lab2` and `lab3` are known in advance and not obtained as a result of the conditional rewrites. A notable example where this alternative encoding cannot be immediately applied are the semantics of process calculi where synchronisation rules do not know in advance which signals are ready to perform their subprocesses.

Another slight improvement is the object-by-object decomposition of object collections instead of the one based on a pair of subsets presented above. For example the above rule scheme becomes:

```
crl {lab1} obj1 conf2 => obj3 conf4
 if {lab2} obj1 => obj3
 /\ {lab3} conf2 => conf4 .
```

A more significant improvement is that in some cases we allow to contextualise rules at any place of a term. We recall that in a SOS derivation this is typically achieved by rules that lift up *silent* (e.g. τ-labelled) actions. Technically this is essentially achieved by declaring as frozen the labelling operator $\{\_\}\_$ only. This allows to apply rules to transform a sub-configuration at any level of the nesting hierarchy. That is, SOS rules like the ones for lifting silent actions across the nesting hierarchy like

```
crl < oid1 : cid1 | c: conf1 , attSet1 > =>
    {tau} < oid1 : cid1 | c: conf2 , attSet1 >
 if conf1 => {tau} conf2 .
```

or rules to lift silent actions among object configurations at the same level of the hierarchy like

```
crl {tau} obj1 conf2 => obj3 conf2
 if {tau} obj1 => obj3 .
```

are not necessary in the SOS* style.

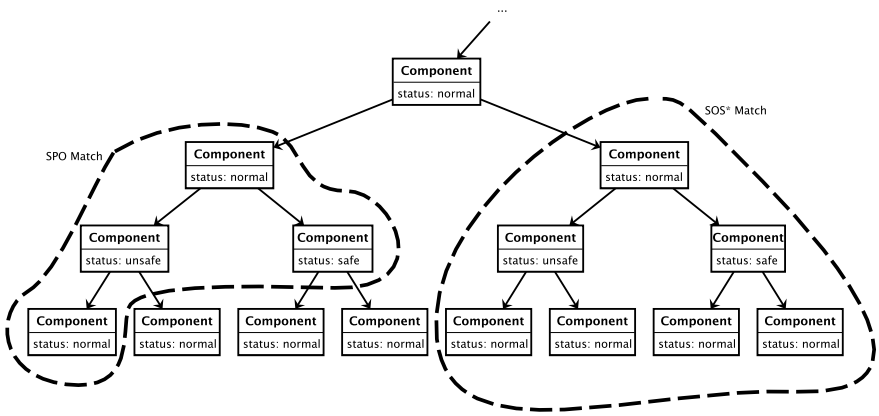

Fig. 1. An instance of the model reconfiguration test case

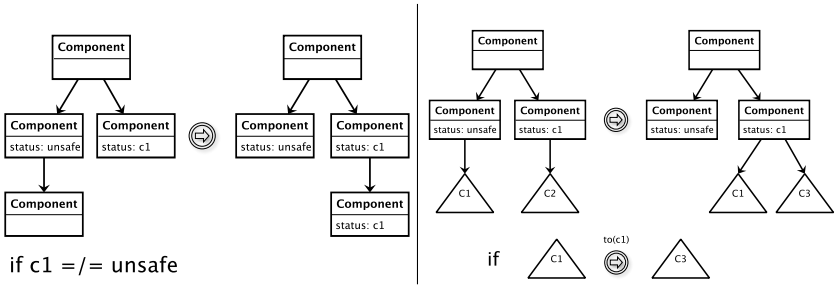

Fig. 2. An SPO rule (left) and a SOS rule (right) for architectural reconfiguration.

4 Benchmark

Our benchmark consists of three test cases selected from the literature as archetypical examples of model reconfigurations, transformations and refactorings. In the following we describe the main features of each test case, emphasizing the most relevant details. For the full description of the test cases we refer to the source code of our implementation [12] or to the indicated references.

Architectural reconfiguration. The first test case we consider is the typical reconfiguration scenario in which some components must be migrated from one compromised location to another one. Many instances of this situation arise in practice (e.g. clients or jobs that must be migrated from one server to another one). Some instances of this scenario can be found e.g. in [7, 13]. In what follows we consider a scenario in which components can be nested within each other. Components within an *unsafe* component `x` must be migrated into an uncle component `y` with the additional requirement of changing their status according

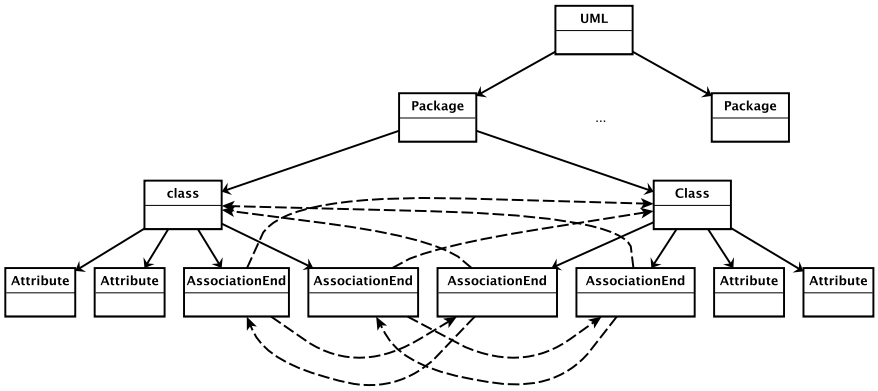

Fig. 3. An instance of the model translation test case

to the status of their new container `y`. Figure 1 depicts one possible instance of the scenario.[3]

The most significant SPO rule[4] is depicted on the left of Fig. 2. It takes an unsafe component and a safe component that are neighbours (they have a common container) and moves the component inside the compromised component one to the safe one while changing its status. More rules are needed (for instance for considering top-level rooms without containers) and some of them have application conditions. As a consequence, the applicability of those rule requires to check the whole model and there is no predefined order on which rules to apply first. The safe system (the system without components in need of evacuation) is reached when no more transformation rules are applicable. For instance, Fig. 1 shows a possible match for the SPO rule. The effect of applying the SPO rule will be to move the normal component under the unsafe one to its new location (under the safe component) while changing its status into safe.

On the right of Fig. 2 instead we the see the main SOS rule: all the components `c1` contained in a unsafe component are evacuated into a safe neighbor component, while changing their status inductively (via `to(c)`-labelled rules). Figure 1 shows a possible instance of the SOS rule. The effect of the SOS rule will be to migrate the two normal components contained in the unsafe component to the safe component while changing their status (in addition the unsafe component is removed).

Model translation. Our second test case is the classical translation of class diagrams into relational database schemes (a description can be found in [14]).

[3] The figures in the paper follow an intuitive UML-like notation, with boxes for objects and arrows for references. We prefer to use this intuitive notation to sketch the scenarios, leaving the detailed Maude implementation [12] for interested readers.

[4] The big encircled arrow separates the rule's left- and right-hand side. Object identifiers are dropped for the sake of clarity and are to be identified by their spatial location.

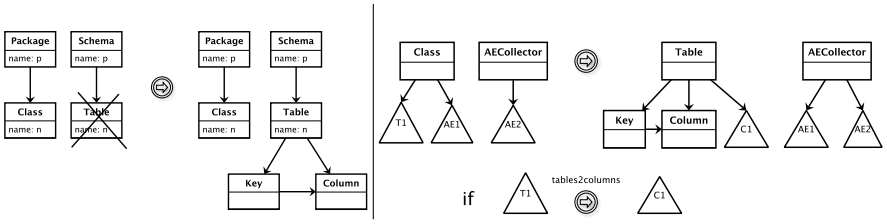

Fig. 4. An SPO rule (left) and a SOS rule (right) for model translation

The main idea is that classes are transformed into tables and their attributes into columns of the tables. Associations between classes are transformed into auxiliary tables with foreign keys from the tables corresponding to the associated classes. Figure 3 depicts one possible instance of the scenario.

Figure 4 sketches two illustrative transformation rules. The SPO rule transforms a class (belonging to a package) into a table (within the corresponding schema). It also creates a primary key and the corresponding column for the table. A negative application condition forbids the application of the rule in case the table already exists. The, let us say, corresponding rule in SOS format transforms a class into a table provided that its attributes are transformed into columns and its association ends are properly collected. An auxiliary object is used as a container where to put association ends of the same relation in the same context so that they can be transformed properly by another rule.

Refactoring. The example of model refactoring we consider is the classical attribute *pull-up* as described in [15]. The main idea is very simple: if all the subclasses of a class `c` declare the same attribute, then the attribute should be declare at `c` only. This preserves the semantics of the diagram (as the sub-classes will inherit the attribute) while simplifying it by removing redundancies. Figure 5 depicts one possible instance of the scenario.

Figure 6 depicts two illustrative transformation rules. The SPO rule pulls an attribute up provided that it is not annotated as *missing* by another class (a set of rules takes care of creating such annotations). The SOS rule instead pulls the attribute up provided that all sibling sub-classes agree to pull it up.

5 Experiments

This section presents our experimental results. Experiments were run on an Ubuntu Linux server equipped with Intel Xeon 2.67GHz processors and 24GB of RAM. Each experiment consists on the transformation of instances of a test case using the discussed representation and transformation styles. Instances are automatically generated with the help of parameterizable instance generators that allow us to, for instance, to scale up the instances to check scalability of the various approaches. For each experiment we have recorded the number of rewrites and the running time, put in the y-axis of separate plots. Each experiment

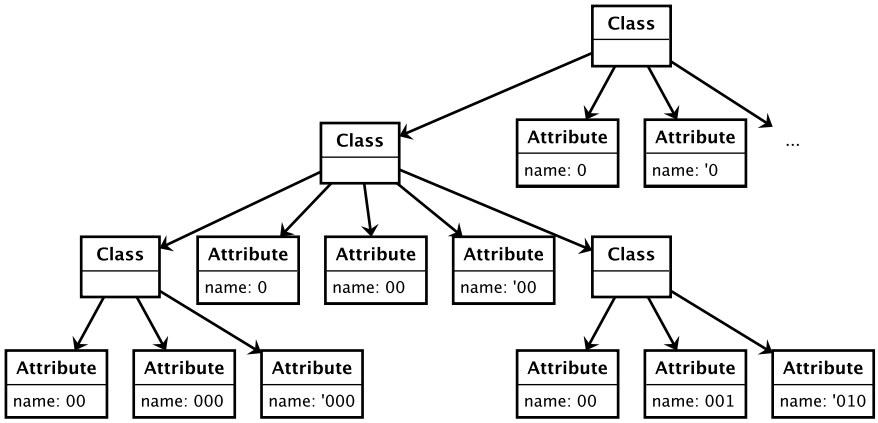

Fig. 5. An instance of the model refactoring test case

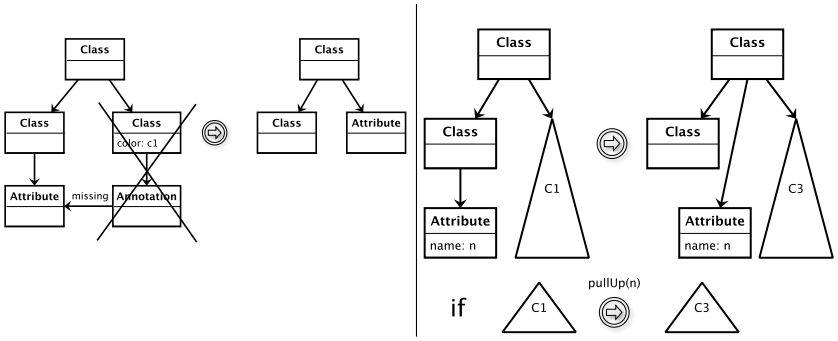

Fig. 6. An SPO rule (left) and a SOS rule (right) for model refactoring

is performed for an increased size factor that typically makes the model grow exponentially. The x-axis corresponds to the size of the instance in terms of overall number of objects. The timeout for the experiments is of an hour. We do not present results for instances larger than those where at least one of the techniques already times out (which is denoted by the interruption of the plot).

The goal of the experiments is to collect evidence of performance differences, draw hypotheses about the causes of those differences and validate our hypotheses with further experiments. Our benchmark consists of the three test cases presented in Section 4. The code for replicating our experiments is available at [12].

5.1 SOS vs SOS*

1st experiment. We start testing the impact of our improvement encoding of SOS (SOS*) with a basic set of instance generators. The generator for the reconfiguration test cases has a single parameter which is the depth of the

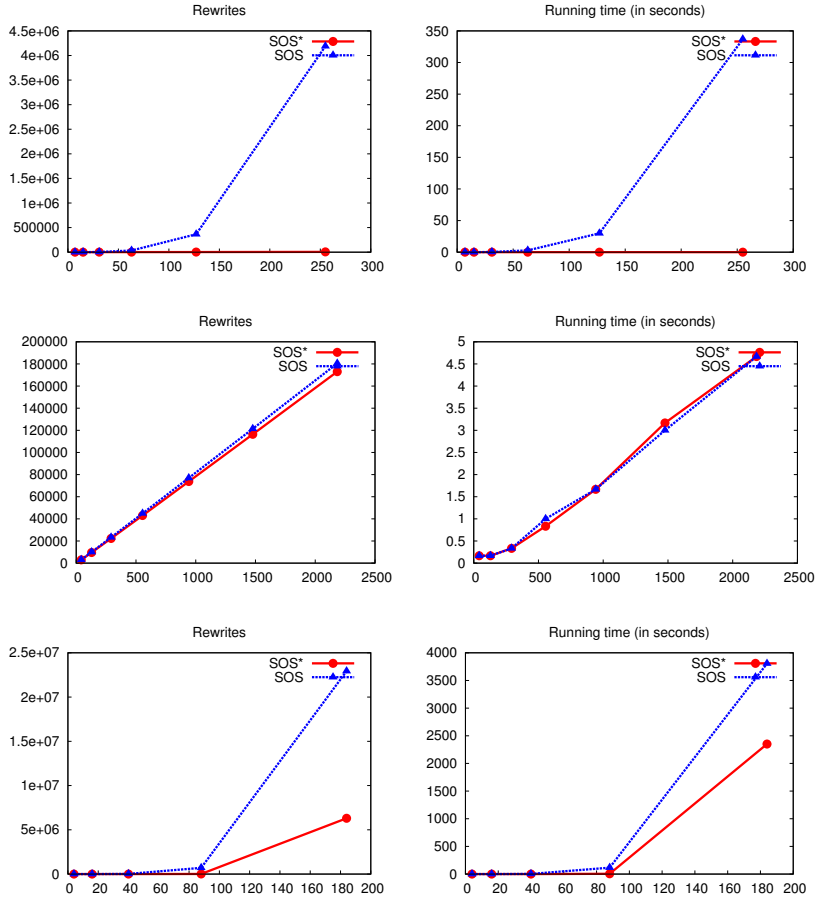

Fig. 7. SOS vs SOS* in reconfiguration (top), translation (middle), pullup (bottom).

component containment tree, i.e. for a given natural number n, it generates a binary tree of depth n. The grandfathers of leafs have exactly one unsafe component and one safe component as children. All other components are normal. Figure 1 sketches one such instance. The parameter of the instance generator for the model transformation case is the branching factor of the containment tree, i.e. given for a given natural number n, it generates a UML domain with n packages, each containing $2n$ classes, each containing n attributes and n associations. The i-th association of class c with c even (resp. odd) has as opposite the i-th association of class $c+1$ (resp. $c-1$). So-built domains have n packages, $2n^2$ classes and n^3 association pairs (c.f. Fig. 3).

The instance generator for the refactoring test case produces binary trees of class hierarchies. Hence, each class has two sub-classes. In addition each sub-class has one local attribute (that will not be pulled up) and one (non-local) attribute

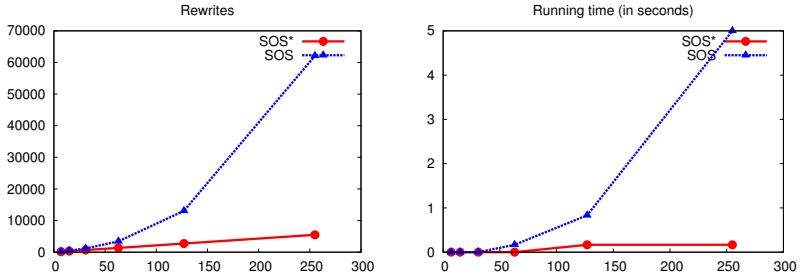

Fig. 8. SOS vs SOS*: effect of disabling new attempts.

inherited from its parent. The topmost class has only one local attribute and one (non-local) attribute (c.f. Fig. 5).

The results of Fig. 7 show a clear superiority of SOS* in most cases. The only exception is the model translation test case. We argue that there are two reasons. First, SOS* allows to contextualise some reconfigurations at an arbitrary level of the nesting hierarchy while SOS have to derive the reconfiguration at the top level by lifting up silent rewrite steps. The second reason is that SOS* performs less transformation attempts as it does not try rules that have unnecessary labels.

The reconfiguration test case is a perfect example for both issues. First, regarding the free contextualisation of top SOS* rules we observe that in the considered instances the rule can be applied at the bottom of the term, while the SOS rules require in addition to lift the application of such rule up to the root. In addition, determining whether a transformation can be carried out can be determined by the non-applicability of rules in the SOS* case, while in the SOS case requires to perform many unsuccessful transformation attempts. In the model translation both styles are essentially equivalent as the top rule must necessarily apply at the top of the term representing the model and after transformation the rules are deactivated as the necessary patterns disappear.

2nd experiment. In order to validate the first hypothesis we have performed experiments where safe components do not accept reconfigurations. In addition a component whose sub-components are safe becomes safe. This does not only disables reconfigurations after possible a migration but also prevents reconfiguration attempts. The results are depicted in Fig. 8 where it can be seen that now SOS scales better since the number of rewrite attempts for silent transitions is reduced (safe components and their containment are not checked for reconfiguration).

3rd experiment. Another improvement of SOS* regards the top-down imposition of labels in rewrite conditions. In order to validate the effect of top-down enacting of transformations we have conducted further experiments with the model reconfiguration test case with a different instance generator: now the root is a normal component, the two sons of the root are a unsafe and a safe component that contain a fixed number components, each able to change into safe plus

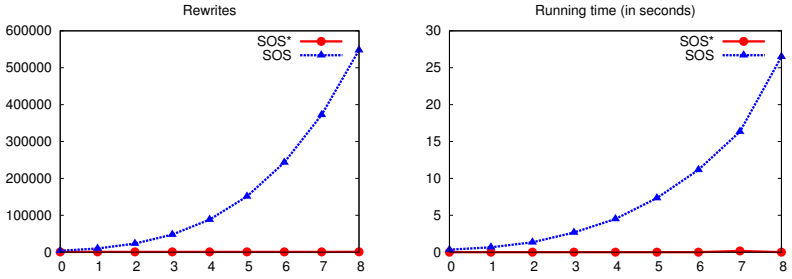

Fig. 9. SOS vs SOS*: effect of increasing the number of enabled actions.

any status of a set of size n, the parameter of the generator. So, for $n = 0$, the components to be migrated are able to change into safe, for $n = 0$ they are ready to change into safe and another status, and so on. The results of such experiment are depicted in Fig. 9.

As expected the SOS* transformation is not affected as n increases. Indeed, the SOS* transformation rules will call for a transformation into safe, while in the SOS transformation all possible status changes will be attempted. As a result the computational effort of SOS transformations blows up with the increase of n.

5.2 SPO vs SOS*

In this set of experiments we compare the SPO approach against the SOS* one.

1st experiment. We start with the first set of instance generators used in §5.1.

The results of Fig. 10 show that SOS* is superior in the reconfiguration test case only. The situation can be roughly explained as follows: matching the migration rule consists on finding a subtree whose root is a component having two subtrees: one having a unsafe component as root and one having a safe one as root. In the SPO case the tree is not parsed: indeed we are given a graph and have to check all possible subsets of nodes to see if they constitute indeed a tree. Instead in the SOS case the tree is already parsed (the parsing is a term of the hierarchical representation) which enormously facilitates rule matching (recall that matching amounts to subgraph isomorphism which is NP-complete). As a consequence, the SPO transformation involves more unsuccessful rule attempts and this is the main reason of the drastic difference in running time (and not in number of effective rewrites).

In the rest of the test cases SPO performs better. This is particularly evident in the refactoring test case where the performance of SOS* degenerates mainly due to the lack of a smart transformation strategy. Indeed it can happen that a pull up has to be attempted at some class every time one of the terms corresponding to one of the subclasses changes. Clearly applying rules bottom up would result in better results but this would require a more cumbersome implementation.

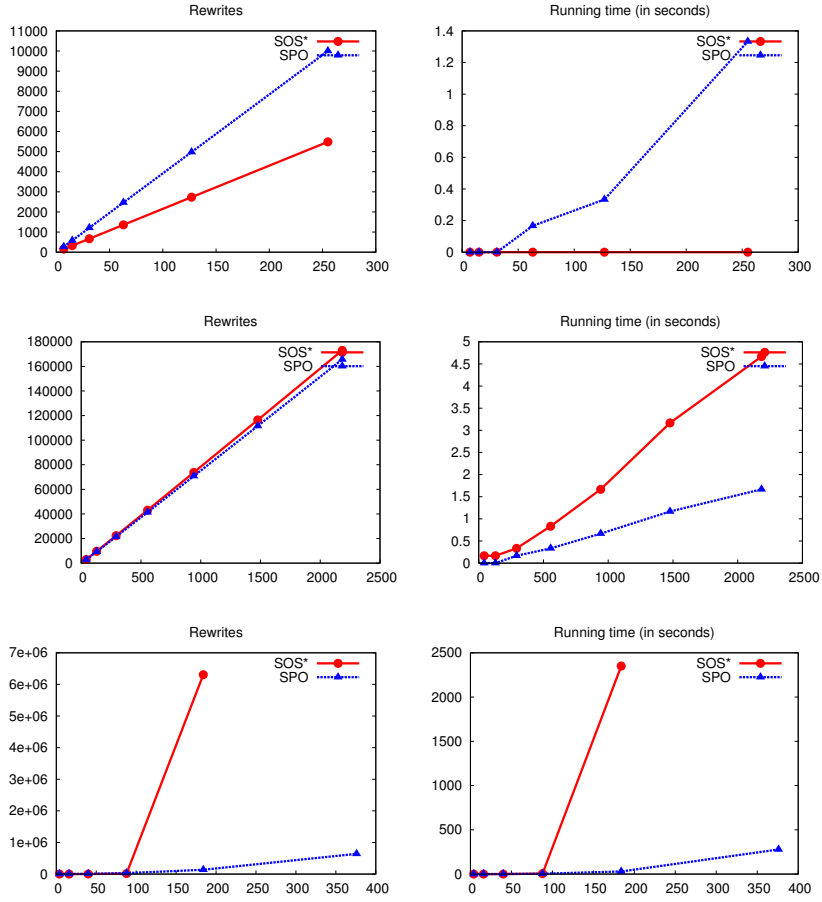

Fig. 10. SPO vs SOS* in reconfiguration (top), translation (middle), pullup (bottom).

We focus now on the transformation test case were we see that SPO performs only slightly better. There are various reasons. First, the structure of the model is rather flat. Indeed the hierarchy is limited to a fixed depth as packages contains classes, classes attributes and associations and that is. So containment trees are of depth 3. In addition, association pairs have to lifted to the top level in the SOS* transformation since the transformation rule that translates them needs them to be in a common context. This involves an overhead that makes SOS* exhibit a worse performance.

2nd experiment. In order to check the impact of such overhead we have performed an additional experiment in which the instances have no associations at all. Figure 11 shows the results were we see how SOS* is the winner this time confirming our hypothesis.

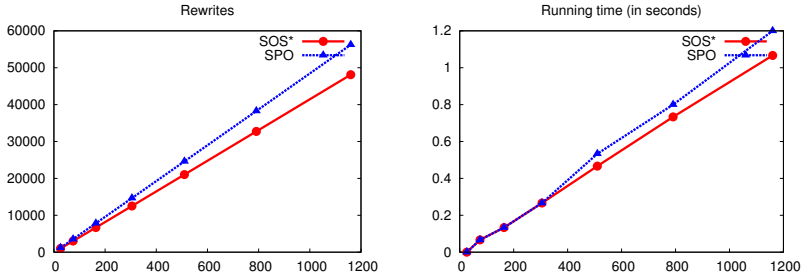

Fig. 11. SPO vs SOS*: effect of removing associations.

6 Conclusion

We have presented an empirical evaluation of the performance of two transformation styles that are very popular in rule-based programming and specification. For instance, in the process algebra community they essentially correspond to the rule formats used for specifying reduction and transition label semantics.

We have focused on model transformations and as a result of our experience we have obtained a set of hints that should be useful for future development of model transformations (or other kind of rule-based specifications) in Maude. We are currently investigating to which extent our experience can be exported to other rule-based frameworks like CafeOBJ [16], Stratego [17] or XSLT [18] with a particular attention to model transformation frameworks such as MOMENT2-MT [19], ATL [20], Stratego/XT [21], and SiTra [22].

It is worth to remark that the aim of the paper is not to compare the performance of transformation tools as done in various works and competitions [23–25]. Rather we assume the point of view of a transformation programmer, which is given a fixed rule-based tool and can only obtain performance gains by adopting the appropriate programming style.

Even if we have focused fundamentally on deterministic transformations many cases (e.g. reconfigurations) are inherently non-deterministic. This gives rise to a state space of possible configurations, whose complexity and required computational effort is influenced again by the chosen rule style.

Another interesting aspect to be investigated is to understand if and how strategy languages (c.f. [26]) or heuristics (c.f. [27]) can be exploited to appropiatedly guide the model transformation process in the most convenient way.

References

1. Meseguer, J.: Conditional rewriting logic as a united model of concurrency. Theoretical Computer Science **96** (1992) 73–155
2. Rozenberg, G., ed.: Handbook of Graph Grammars. World Scientific (1997)
3. Bruni, R., Lluch Lafuente, A., Montanari, U.: On structured model-driven transformations. International Journal of Software and Informatics (IJSI) **2** (2011)

4. Boronat, A., Bruni, R., Lluch-Lafuente, A., Montanari, U., Paolillo, G.: Exploiting the hierarchical structure of rule-based specifications for decision planning. In: International Joint Conference on Formal Techniques for Distributed Systems (FMOODS/FORTE'10). Volume 6117 of LNCS., Springer (2010)
5. Meseguer, J., Talcott, C.: Semantic models for distributed object reflection. In Magnusson, B., ed.: 16th European Conference on Object-Oriented Programming (ECOOP'02). Volume 2374 of LNCS. Springer (2006) 1637–1788
6. Meseguer, J.: A logical theory of concurrent objects. In: International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA/ECOOP'90). (1990) 101–115
7. Boronat, A., Meseguer, J.: An algebraic semantics for MOF. In: International Conference on Fundamental Aspects of Software Engineering (FASE'08). Volume 4961 of LNCS., Springer (2008) 377–391
8. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L.: All About Maude. Volume 4350 of LNCS. Springer (2007)
9. Plotkin, G.D.: A structural approach to operational semantics. Journal of Logic and Algebraic Programming **60-61** (2004) 17–139
10. Verdejo, A., Martí-Oliet, N.: Executable structural operational semantics in Maude. Journal of Logic and Algebraic Programming **67** (2006) 226–293
11. Bruni, R., Meseguer, J.: Semantic foundations for generalized rewrite theories. Theoretical Computer Science **360** (2006) 386–414
12. `http://cse.lab.imtlucca.it/~lluch/facs2011.tgz`.
13. Bruni, R., Lluch Lafuente, A., Montanari, U., Tuosto, E.: Style based architectural reconfigurations. Bulletin of the European Association of Theoretical Computer Science (EATCS) **94** (2008) 161–180
14. Boronat, A., Knapp, A., Meseguer, J., Wirsing, M.: What is a Multi-Modeling Language? In: WADT'08. Volume 5486 of LNCS., Springer (2009) 71–87
15. Biermann, E., Ehrig, K., Köhler, C., Kuhns, G., Taentzer, G., Weiss, E.: EMF model refactoring based on graph transformation concepts. In: 3rd Workshop on Software Evolution through Transformations. Volume 3., ECEASST (2006)
16. CafeObj, `http://www.ldl.jaist.ac.jp/cafeobj/`.
17. Stratego, `http://www.program-transformation.org/Stratego/`.
18. XSLT, `http://www.w3.org/TR/xslt20/`.
19. MOMENT2-MT: `www.cs.le.ac.uk/people/aboronat/tools/moment2-gt/`.
20. ATL, `http://www.eclipse.org/atl/`.
21. StrategoXT, `http://strategoxt.org/`.
22. SiTra, `http://www.cs.bham.ac.uk/~bxb/SiTra.html`.
23. Rewrite engines competition, `www.lcc.uma.es/~duran/rewriting_competition/`.
24. Graph Transformation Contest, `http://fots.ua.ac.be/events/grabats2008/`.
25. Varró, G., Schürr, A., Varró, D.: Benchmarking for graph transformation. In: IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05), IEEE Computer Society (2005) 79–88
26. Braga, C., Verdejo, A.: Modular structural operational semantics with strategies. ENTCS **175** (2007) 3–17
27. Kessentini, M., Sahraoui, H., Boukadoum, M., Omar, O.: Search-based model transformation by example. Software and Systems Modeling (2010) 1–18